Towards a Distributed Search Engine

Ricardo Baeza-Yates

Yahoo! Research Barcelona, Spain

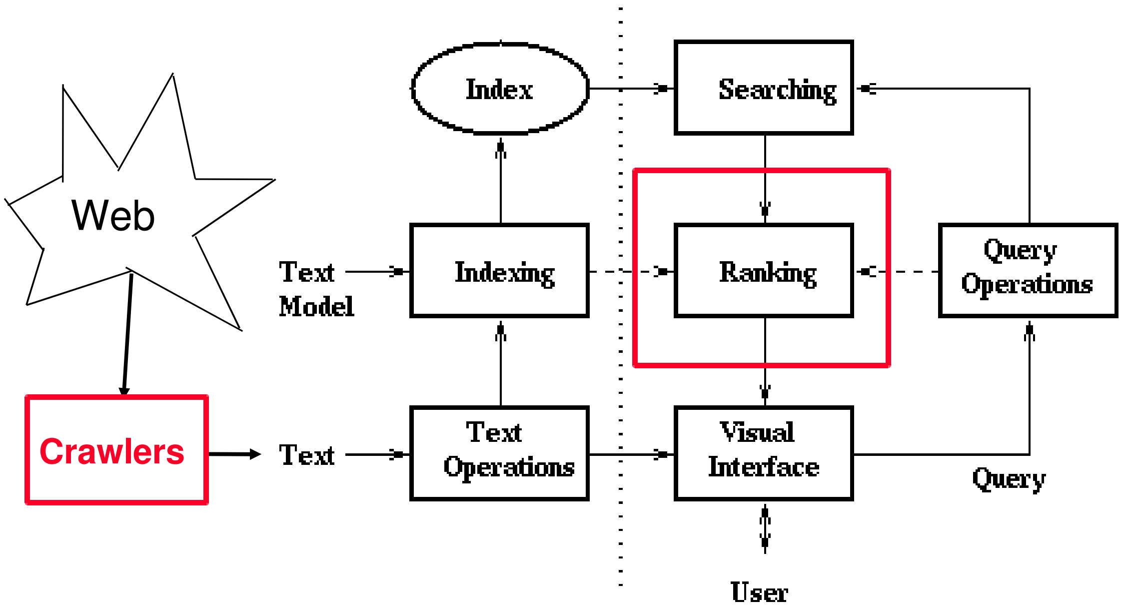
Web Search

- This is one of the most complex data engineering challenges today:
  - Distributed in nature
  - Large volume of data
  - Highly concurrent service
  - Users expect very good & fast answers
- Current solution: Centralized system

INFOSCALE 2008, June 4-6, Vico Equense, Italy

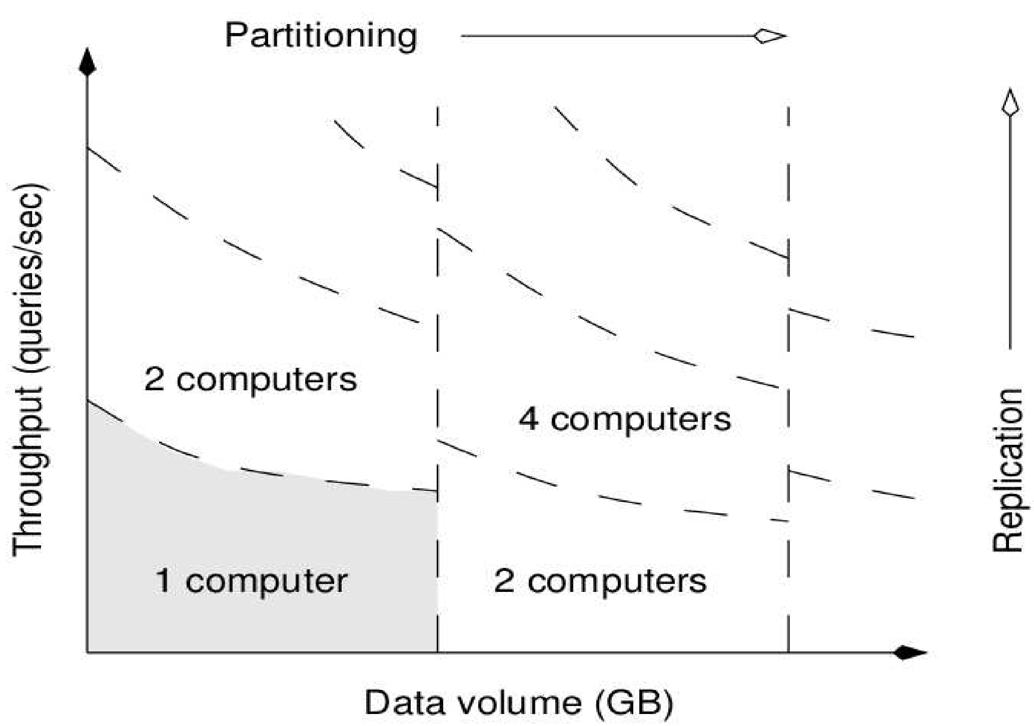
978-963-9799-28-8
DOI 10.4108/infoscale.2008.31

WR System Architecture



Scaling Up

From [Moffat and Zobel, 2004]



Inverted Index

Dictionary	Inverted Lists
cold	<2,1> <4,1> <5,1>
hot	<1,1> <4,1> <5,1> <6,1>
in	<3,1> <6,1>
not	<4,1> <5,1>
pease	<1,1> <2,1> <3,1> <4,2> <5,2> <6,1>
porridge	<1,1> <2,1> <3,1> <4,2> <5,2> <6,1>
pot	<3,1> <6,1>
the	<3,1> <6,1>

System Size

- 20 billion Web pages implies at least 100Tb of text
- The index in RAM implies at least a cluster of 3,000 PCs
- Assume we can answer 1,000 queries/sec
- 73 million queries a day imply 2,000 queries/sec
- Decide that the peak load plus a fault tolerance margin is 5
- This implies a replication factor of 10 giving 30,000 PCs
- Total deployment cost of over 100 million US$ plus maintenance cost
- In 2010, being conservative, we would need over 1 million computers!

Questions

- Should we use a centralized system?
- Can we have a (cheaper) distributed search system in spite of network latency?

- Preliminary answer: Yes
- Solutions: caching, pruned indexes, new ways of partitioning the index, exploit locality when processing queries, etc.

Advantages

- Distribution decreases replication, crawling, and indexing and hence the cost per query
- We can exploit high concurrency and locality of queries
- We can also exploit the network topology
- Main design problems:
  - Depends upon many external factors that are seldom independent
  - One poor design choice can affect performance or/and costs

Challenges

- Must return high quality results (handle quality diversity and fight spam)
- Must be fast (fraction of a second)
- Must have high capacity
- Must be dependable (reliability, availability, safety and security)
- Must be scalable

Caching

- Caching can save significant amounts of computational resources
  - Search engine with capacity of 1000 queries/second
  - Cache with 30% hit ratio increases capacity to 1400 queries/second
- Caching helps to make queries “local”
- Caching is similar to replication on demand

Caching basics

- A cache is characterized by its size and its eviction policy
- *Hit* : requested item is already in the cache
- *Miss* : requested item is not in the cache

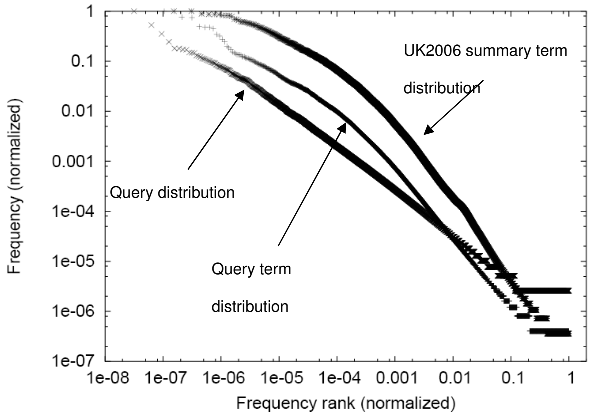
- Caches speed up access to frequently or recently used data
  - Memory pages, disk, resources in LAN / WAN

Caching in Web Search Engines

- Caching **query results** *versus* caching **posting lists**
- **Static** *versus* **dynamic** caching policies
- Memory allocation between different caches
- Baeza-Yates et al, SIGIR 2007

Data characterization

- 1 year of queries from Yahoo! UK
- UK2006 summary collection
- Pearson correlation between query term frequency and document frequency = 0.424



Caching query results or term postings

- Queries
  - 50% of queries are unique
  - 44% of queries are singleton (appear only once)
  - Infinite cache achieves 50% hit-ratio
    - Infinite hit ratio = (#queries – #unique) / #queries

- Query terms
  - 5% of terms are unique (the vocabulary)
  - 4% of terms are singleton
  - Infinite cache achieves 95% hit ratio

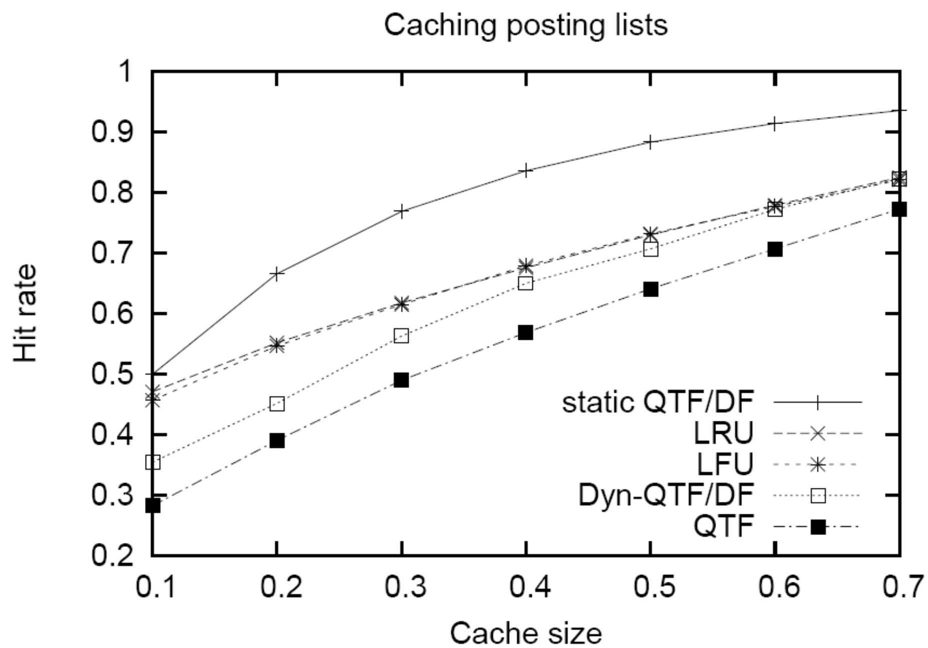
Static Caching of Postings

- QTF for static caching of postings (Baeza-Yates & Saint-Jean, 2003):
  - Cache postings of terms with the highest $f_q(t)$
- Tradeoff between $f_q(t)$ and $f_d(t)$
  - Terms with high $f_q(t)$ are good to cache
  - Terms with high $f_d(t)$ occupy too much space
- QTFDF: Static caching of postings
  - Knapsack problem:
  - Cache postings of terms with the highest $f_q(t)/f_d(t)$

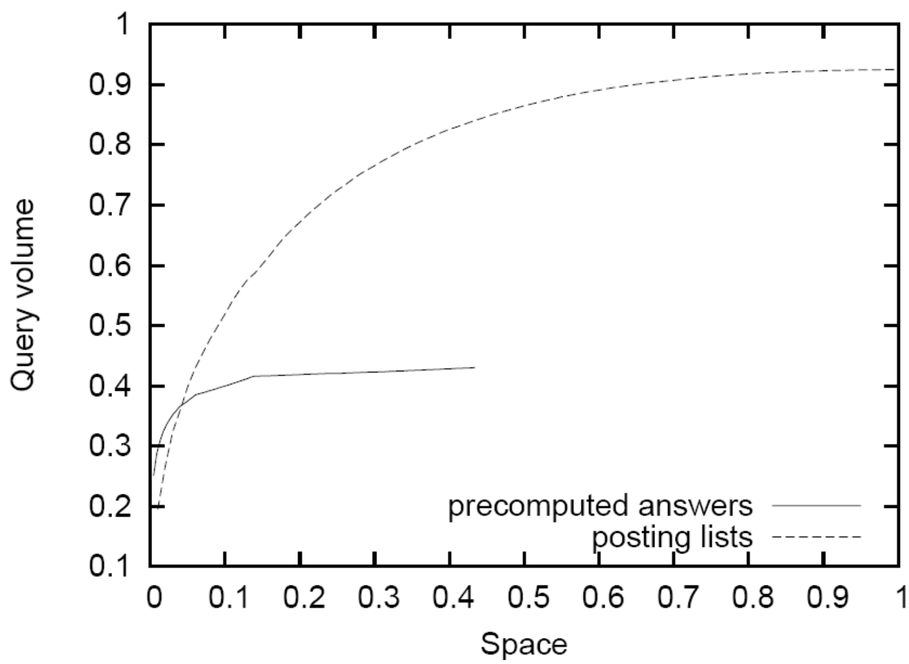
Evaluating Caching of Postings

- Static caching:
  - QTF : Cache terms with the highest query log frequency $f_q(t)$
  - QTFDF : Cache terms with the highest ratio $f_q(t)$ / $f_d(t)$
- Dynamic caching, we employ:
  - LRU, LFU
  - Dynamic QTFDF : Evict the postings of the term with the lowest ratio $f_q(t)$ / $f_d(t)$

Results



Combining caches of query results and term postings



Experimental Setting

- Process 100K queries on the UK2006 summary collection with Terrier
- Centralized IR system
  - Uncompressed/compressed posting lists
  - Full/partial query evaluation
- Model of a distributed retrieval system
  - broker communicates with query servers over LAN or WAN

Parameter Estimation

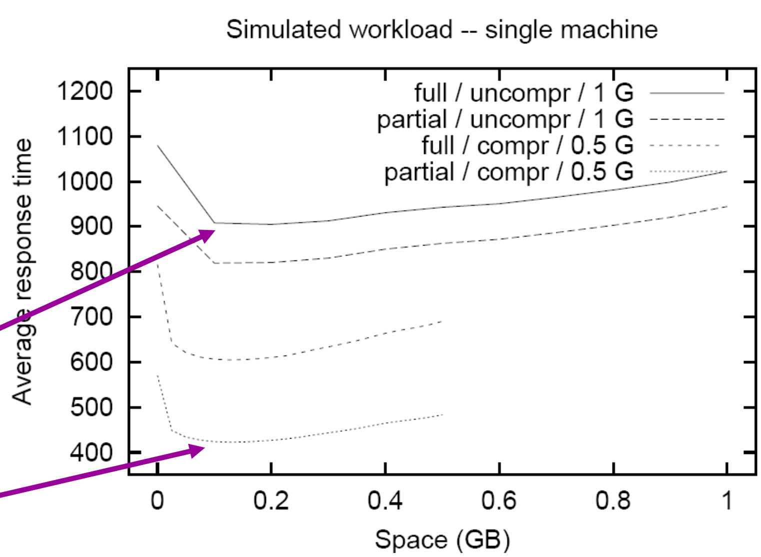
- The average ratio between the time to return an answer computed from posting lists and from the query result cache is:
  - TR_1 : when postings are in memory
  - TR_2 : when postings are on disk
  - M is the cache size in answer units
    - A cache of query results stores $N_c=M$ queries
  - L is the average posting list size
    - A cache of postings stores $N_p=M/L= N_c/L$ posting lists

Parameter Values

	Uncompressed Postings (L=0.75)		Compressed Postings (L'=0.26)	
Centralized system	TR_1	TR_2	TR_1'	TR_2'
Full evaluation	233	1760	707	1140
Partial evaluation	99	1626	493	798
WAN system	TR_1	TR_2	TR_1'	TR_2'
Full evaluation	5001	6528	5475	5908
Partial evaluation	4867	6394	5270	5575

Centralized System Simulation

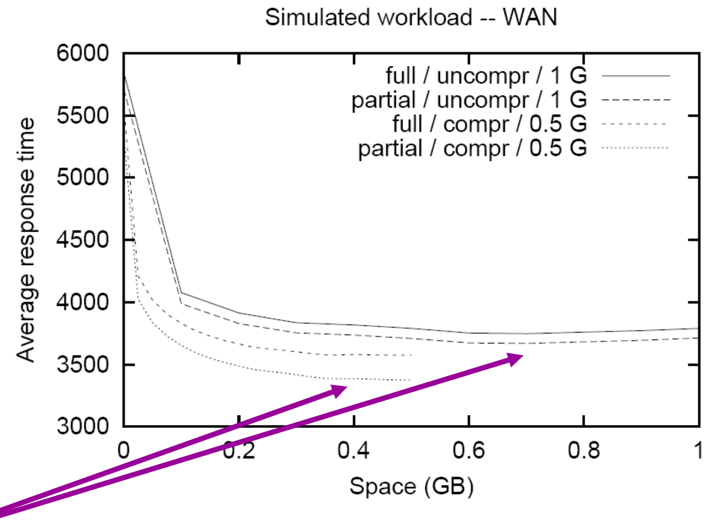
- Assume M memory units
  - x memory units for static cache of query results
  - M-x memory units for static cache of postings
- Full query evaluation with uncompressed postings
  - 15% of M for caching query results
- Partial query evaluation with compressed postings
  - 30% of M for caching query results



WAN System Simulation

- Distributed search engine
  - Broker holds query results cache
  - Query processors hold posting list cache

- Optimal Response time is achieved when most of the memory is used for caching answers



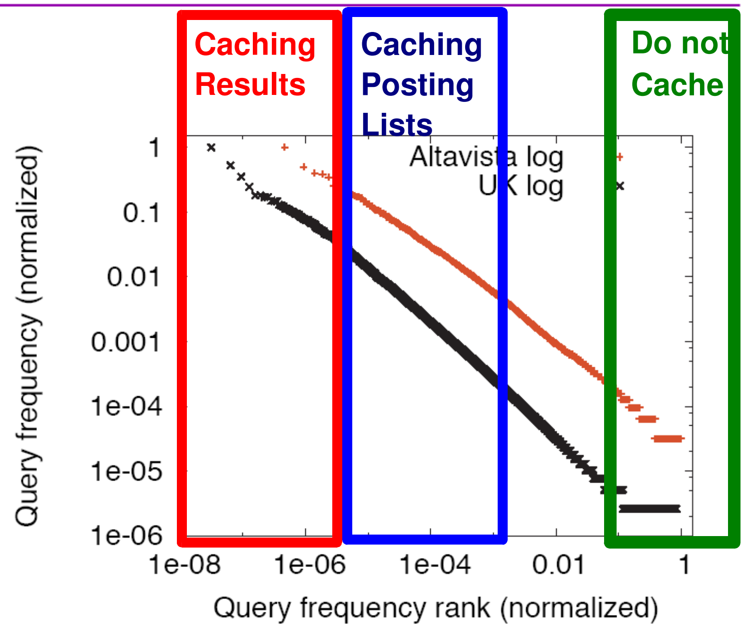
Query dynamics

- Static caching of query results
  - Distribution of queries change slowly
  - A static cache of query results achieves high hit rate even after a week

- Static caching of posting lists
  - Hit rate decreases by less than 2% when training on 15, 6, or 3 weeks
  - Query term distribution exhibits very high correlation (>99.5%) across periods of 3 weeks

Why caching results can't reach high hit rates

- AltaVista: 1 week from September 2001
- Yahoo! UK: 1 year
  - Similar query length in words and characters
- Power-law frequency distribution
  - Many infrequent queries and even singleton queries
- No hits from singleton queries



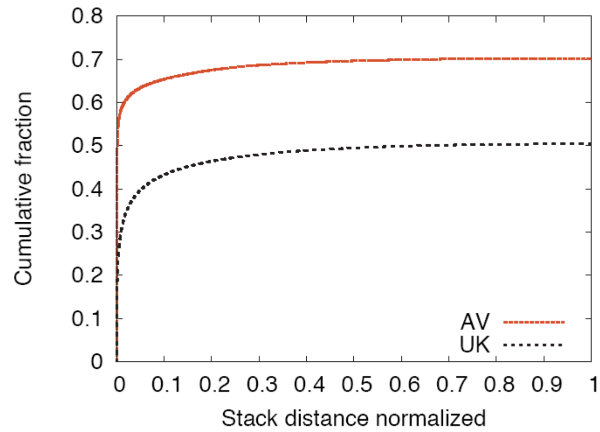
Benefits of filtering out infrequent queries

- Optimal policy does not cache singleton queries
- Important improvements in cache hit ratios

Cache size	Optimal		LRU	
	AV	UK	AV	UK
50k	**67.49**	**32.46**	59.97	17.58
100k	**69.23**	**36.36**	62.24	21.08
250k	**70.21**	**41.34**	65.14	26.65

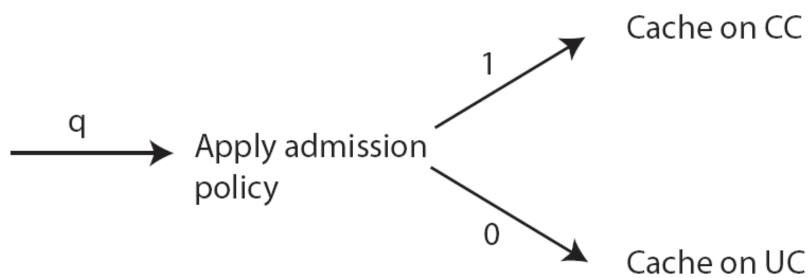
Temporal locality across different query logs

- Temporal locality
  - Stack distance between consecutive occurrences
- More locality
  - Higher hit rate
- AltaVista presents significantly more locality



Admission Controlled Cache (AC)

- General framework for modelling a range of cache policies



- Split cache in two parts
  - Controlled cache (CC)
  - Uncontrolled cache (UC)
- Decide if a query q is frequent enough
  - If yes, cache on CC
  - Otherwise, cache on UC

Baeza-Yates et al, SPIRE 2007

Why an uncontrolled cache?

- Deal with errors in the predictive part
- Burst of new frequent queries
- Open challenge:
  - How the memory is split in both types of cache?

Features for admission policy

- Stateless features
  - Do not require additional memory
  - Based on a function that we evaluate over the query
  - Example: query length in characters/terms
    - Cache on CC if query length < threshold
- Stateful features
  - Uses more memory to enable admission control
  - Example: past frequency
    - Cache on CC if its past frequency > threshold
    - Requires only **a fraction** of the memory used by the cache

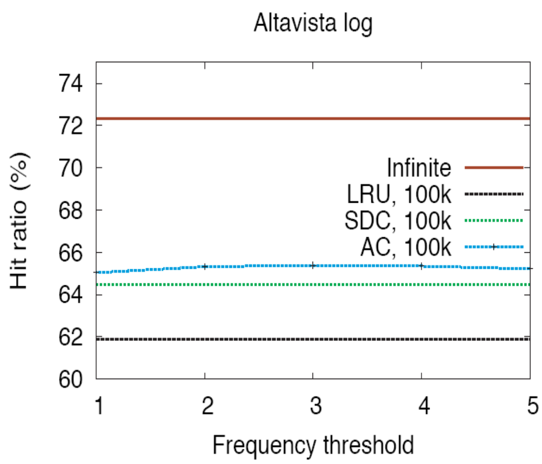
Evaluation

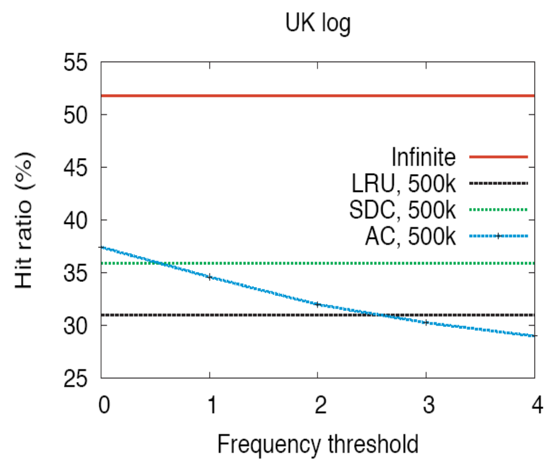
- AltaVista and Yahoo! UK query logs
- Query logs split into 2 parts
  - First 4.8 million queries for training
  - Testing on the rest of the queries
- Compare AC with
  - LRU
  - SDC

LRU and SDC policies

- Eviction policies
  - Once the cache is full, decide which query to evict
- LRU : Evicts the least recent query results
- SDC : Splits cache into two parts
  - Static: filled up with most frequent past queries
  - Dynamic: uses LRU

Results for Stateful Features





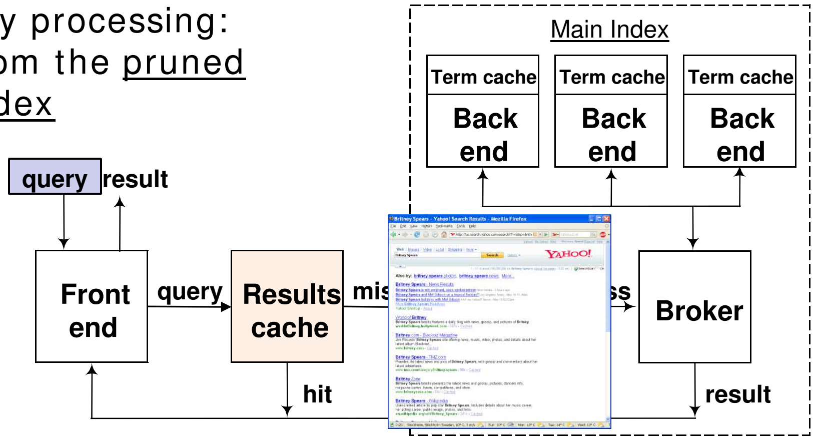
Results for Stateless features

- AC with stateless features outperforms LRU
- Stateless features offer high recall but low precision

	AV		UK	
Infinite	72.32		51.78	
Sizes	50k	100k	100k	500k
LRU	59.49	61.88	21.03	30.96
SDC	**62.25**	**64.49**	**29.61**	**35.91**
AC k_c=10	60.01	59.53	17.07	27.33
AC k_c=20	58.05	62.36	22.85	32.35
AC k_c=30	56.73	61.91	21.60	31.06
AC k_c=40	56.39	61.68	21.19	30.53
AC k_w=2	59.92	62.33	23.10	32.50
AC k_w=3	59.55	61.96	21.94	31.47
AC k_w=4	59.18	61.60	21.16	30.51
AC k_w=5	59.01	61.43	20.81	30.02

Index Pruning (Skobeltsyn et al, SIGIR08)

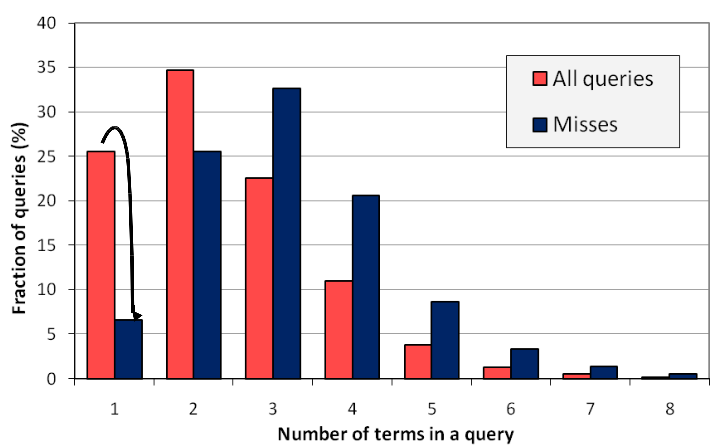
Query processing:
3. from the pruned index



- Results Caching and Index Pruning together
- … to reduce **latency** and **load** on back-end servers

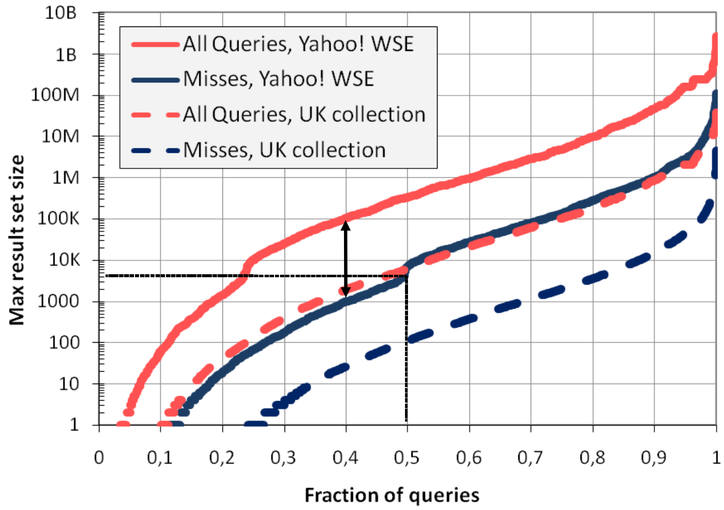
All queries vs. *Misses*: Number of terms in a query

- Average number of terms for *all queries* = **2.4**, for *misses* = **3.2**
- Most single term queries are hits in the results cache
- Queries with many terms are unlikely to be hits



All queries vs. *Misses*: Query result size distribution

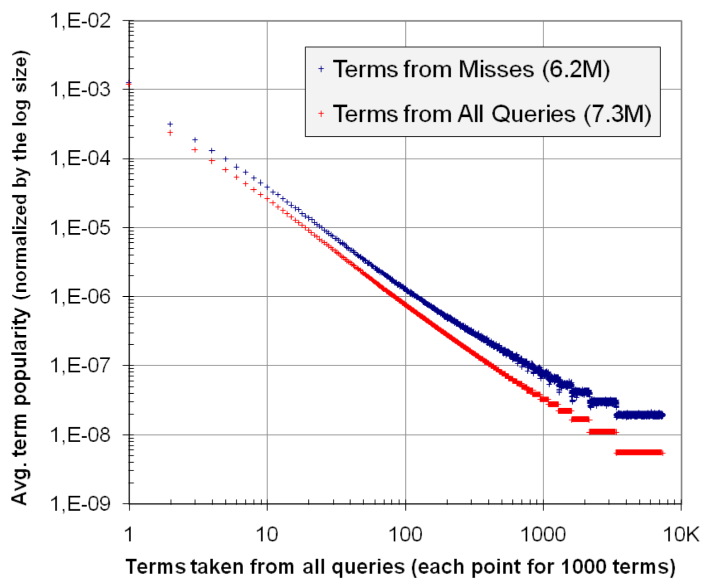
- Randomly selected **2000** queries from *all queries* and *misses*:
- Avg. result size for *misses* is ~**100** times smaller than for *all queries*
- Approx. half of the *misses* returns less than **5000** results – SMALL!
- Similar results with a “small” UK document collection (78M)



All queries vs. *Misses*: Term popularity distribution

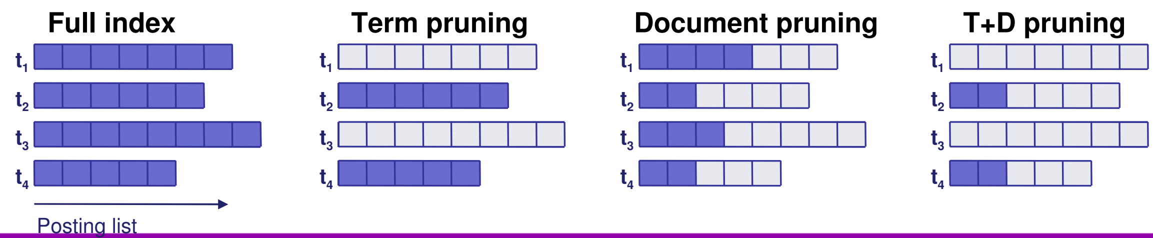
- Each point -> avg. popularity of **1000** consecutive terms
- Popularity is normalized by the size of the log
- The order of terms for *misses* is the same as for *all queries*
- Term popularity **does not** change much!

Log sizes: 185M – *all queries*, 41M - *misses*



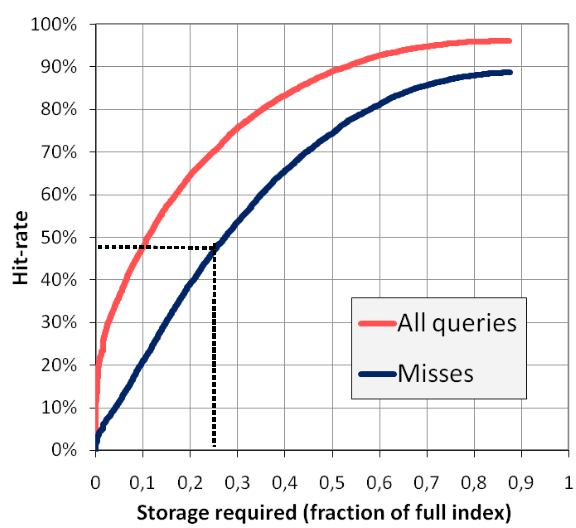
Static index pruning

- Smaller version of the main index, returns:
  - the top-*k* response that is *the same* to the main index's, or
  - a *miss* otherwise.
- Assumes Boolean query processing
- Types of pruning:
  - **Term pruning** – full posting lists for selected terms
  - **Document pruning** – prefixes of posting lists
  - **Term+Document pruning** – combination of both



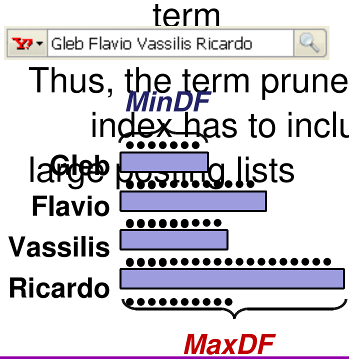
Term Pruning: Performance

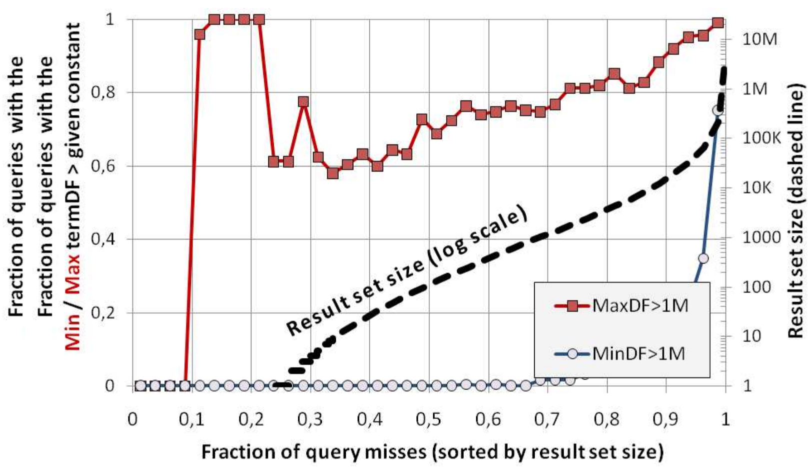
- Answers a query if **all** query terms are in the pruned index
- UK document collection - **78M** documents
- Term pruning based on *profit(t)=popularity(t)/df(t)*
- Performs well for *all queries*
- For *misses* as well: e.g., can process almost **50%** of the queries with **25%** of the index



Term pruning: Frequent terms in *misses*

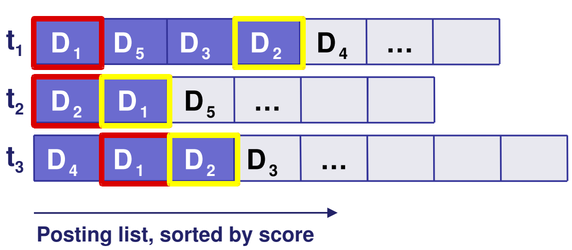
- *Misses* are sorted by the result set size (dashed line)
- *MaxDF* (df of the most frequent query term) is **high** for most of the misses
  MinDF (df of the least frequent query term) **correlates** to the result size
- Many *misses* contain at least one freque term
- Thus, the term pruned index has to incluc large posting lists





Document pruning

- Based on Fagin's top-*k* intersection algorithm
- Keeps postings with high scores only:
  - Sufficient to compute top-*k* results for some queries
- Determining correctness of the result requires computing of a scoring threshold – LATENCY!



Top-2 results:

D_1 D_2

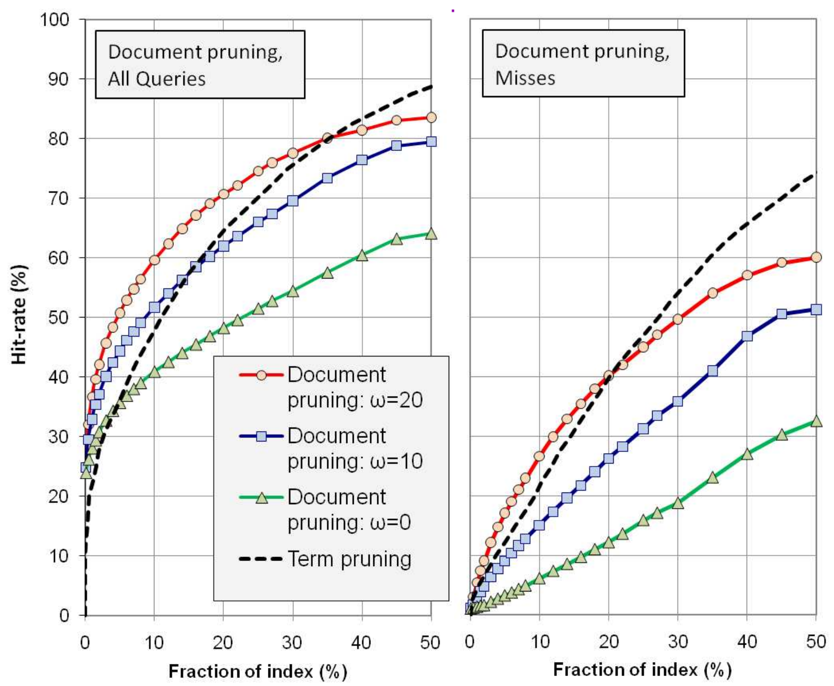
Score threshold:
$s(D_2,t_1)+s(D_1,t_2)+s(D_2,t_3)$

Document pruning: Experimental setup

- Scoring function: $score(d,q) = \sum_{\forall t \in q} \left(bm25(t,d) + \omega \frac{pr(d)}{pr(d)+k} \right)$
  - ***pr(d)*** – query independent score of the document d (pagerank)
  - ω, ***k*** – normalization constants:
    - ω=[0,10,20]
    - *k=1*
- We only look at the **upper bound** for the hit rate:
  - Whether the original top-10 results found in the top portions of all PLs?

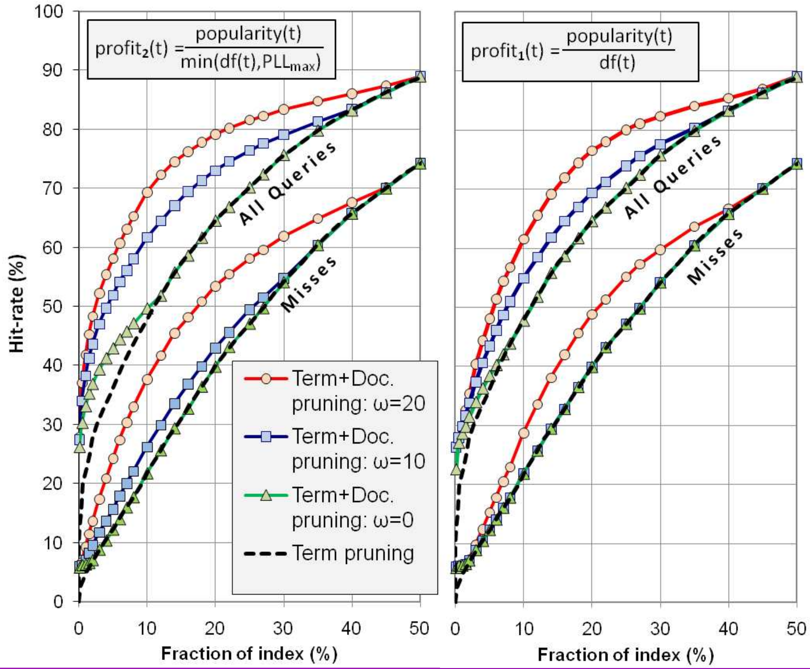
Document pruning: performance

- Doc. pruning needs high weights of pagerank to outperform term pruning, especially for *misses*



Term+Document pruning: performance

- T+D pruning is the best but expensive (high latency)
- *profit₂* is better than *profit₁*
- However, the improvement is marginal for *misses* (with high pagerank weights only)



Analysis of results

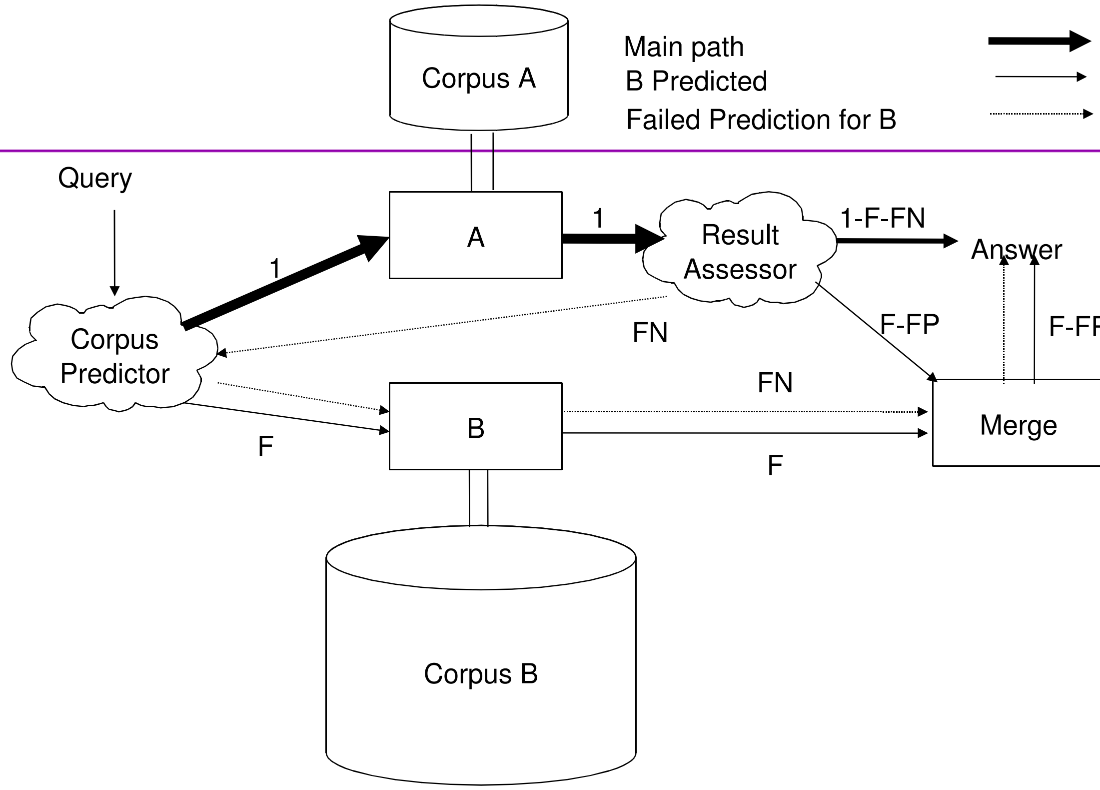
- **Static index pruning**: addition to results caching, not replacement
  - **Term pruning** performs well for *misses* also
    => can be combined with results cache
  - **Document pruning** performs well for *all queries,* but requires high pagerank weights with *misses*
  - **Term+Document pruning** improves over document pruning, but has the same disadvantages
- **Pruned index** grows with collection size
- Document **pruning** targets the same queries as **result caching**
- **Lesson learned:** Important to consider the interaction between the components

Locality

- Many queries are local
  - The answer returns only local documents
  - The user clicks only on local documents
- Locality also helps in:
  - Latency of HTTP requests (queries, crawlers)
  - Personalizing answers and ads
- Can we decrease the cost of the search engine?

Tier Prediction (Baeza-Yates et al, 2008)

- Can we predict if the query is local?
  - Without looking at results
  - or increasing the extra load in the next level
- This is also useful in centralized search engines
  - Multiple tiers divided by quality
- Experimental results for
  - WT10G and UK/Chile collections



Experimental Results

- Centralized case:

	Random	Centralized
Classifier Accuracy	0.714 ±0.008	0.789±0.009
Precision	n/a	0.983±0.006
Recall	na	0.265±0.022

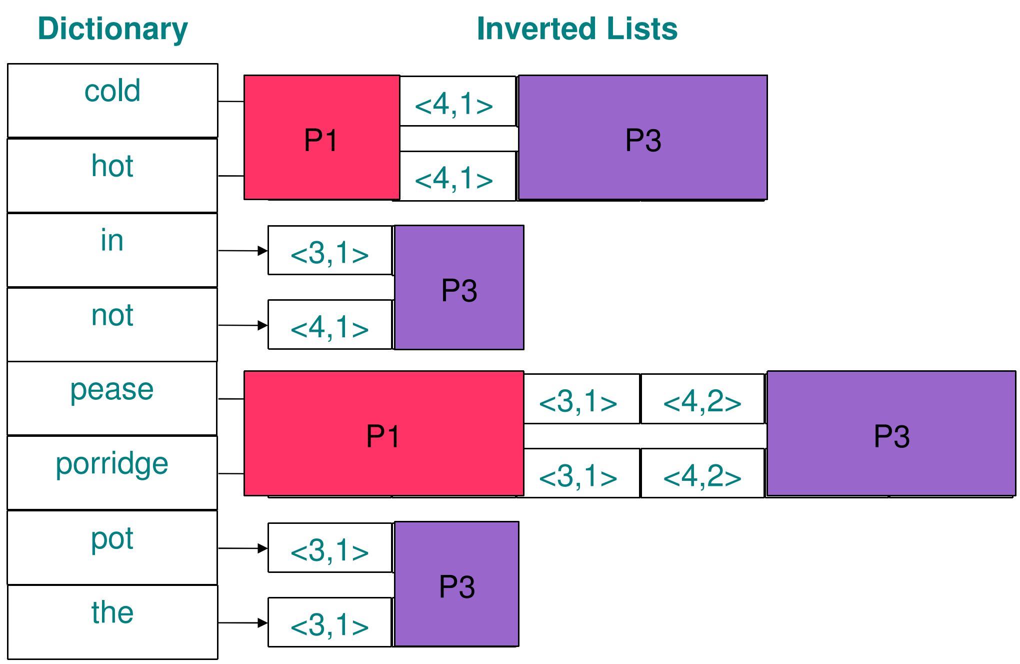
- Distributed case:

	Random	Distributed
Classifier Accuracy	0.539 ±0.006	0.776±0.006
Precision	n/a	0.675±0.006
Recall	n/a	0.991±0.003

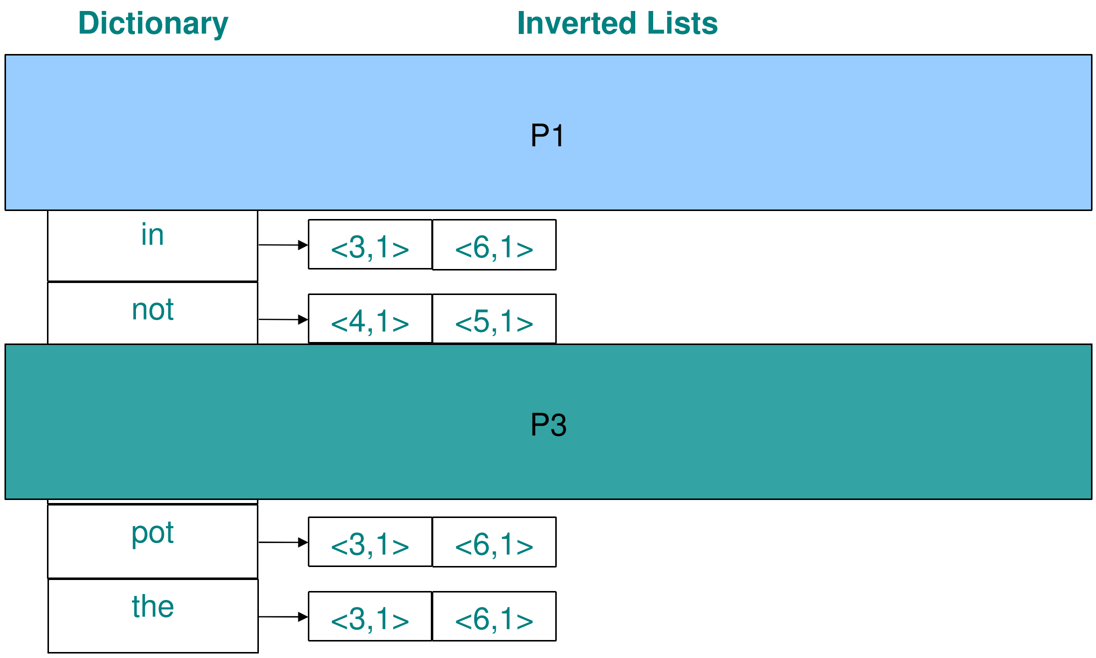
Tier Prediction Example

- Example:
  - System A is twice faster than System B
  - System B costs twice the costs of System A
- Centralized case:
  - 29% answer time improvement at 31% extra cost
- Distributed case:
  - 12% answer time improvement at 18% extra cost

Document Partitioning



Term Partitioning

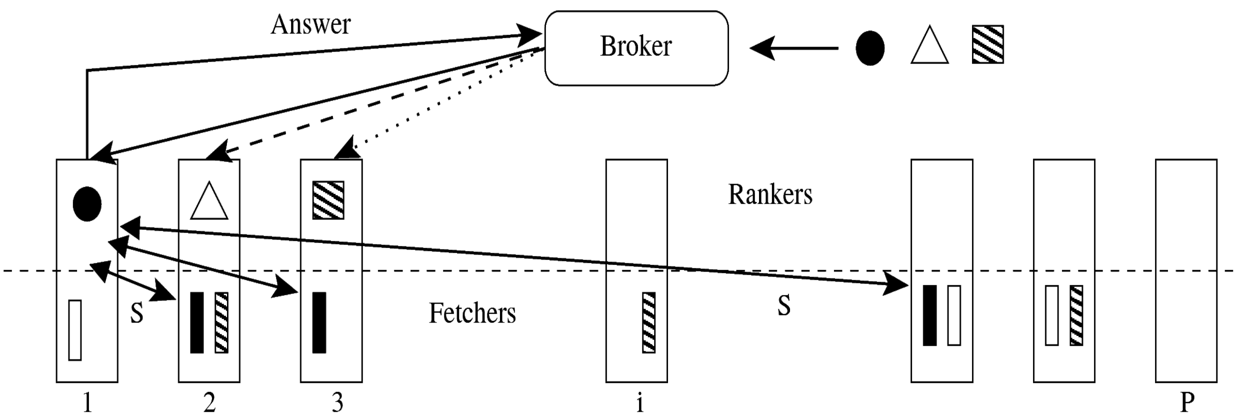


Partitioning the Indexing

- By documents
- Easy to partition
- **Easier to build**
- No concurrency
- Perfect balance
- Less variance
- **Easier to maintain**

- By terms
- Random partition
- Hard to build
- Concurrent
- Less balanced
- Higher variance
- Harder to maintain

Query Processing: Round Robin



Case of term partitioning

Marin et al, 2008

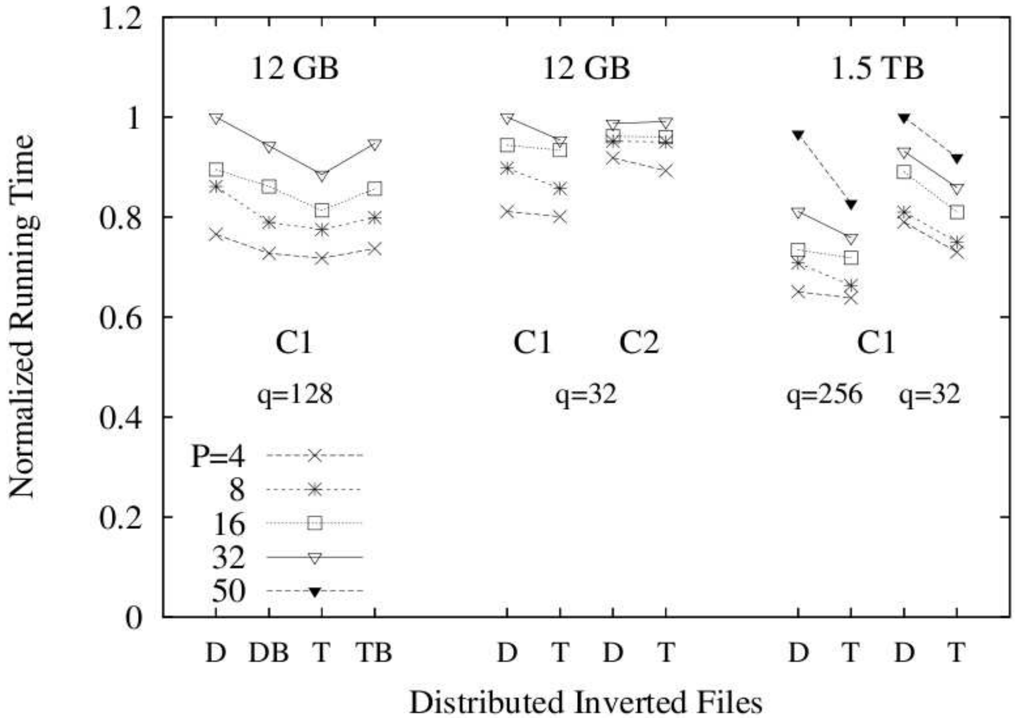
Analysis

- BSP model
- Super-steps + synchronization

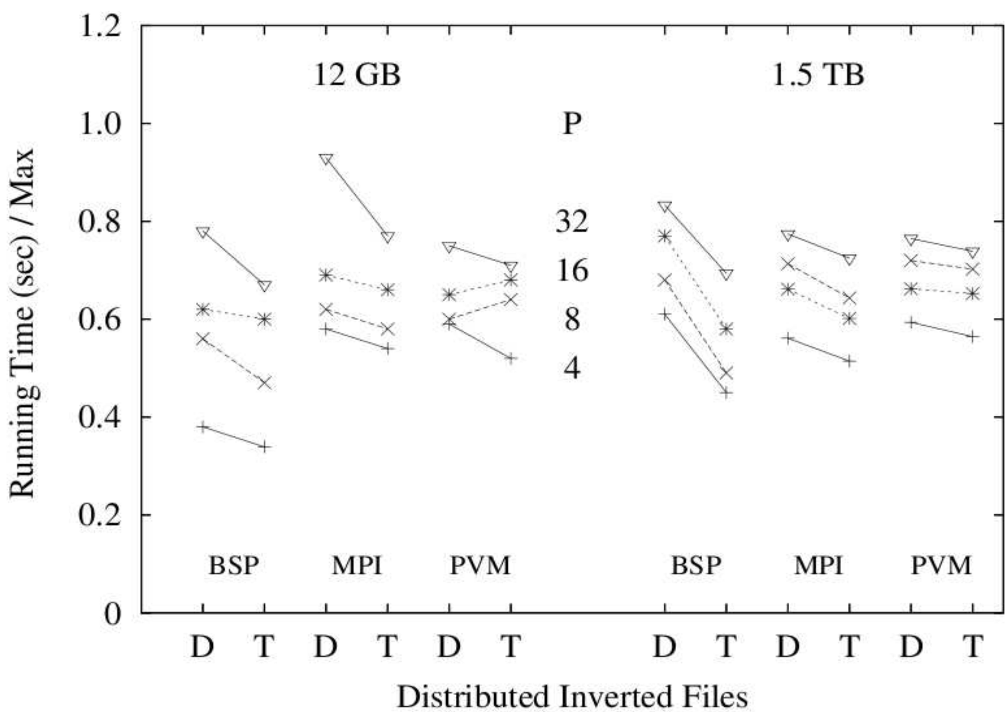
$$t_D = r\,K\,D\,/\,P \,+\, r\,G\,K\,/\,P \,+\, r\,\mathrm{Rank}(K) \,+\, L$$

$$t_T = r\,K\,D \,+\, r\,G\,K \,+\, r\,\mathrm{Rank}(K) \,+\, L\,.$$

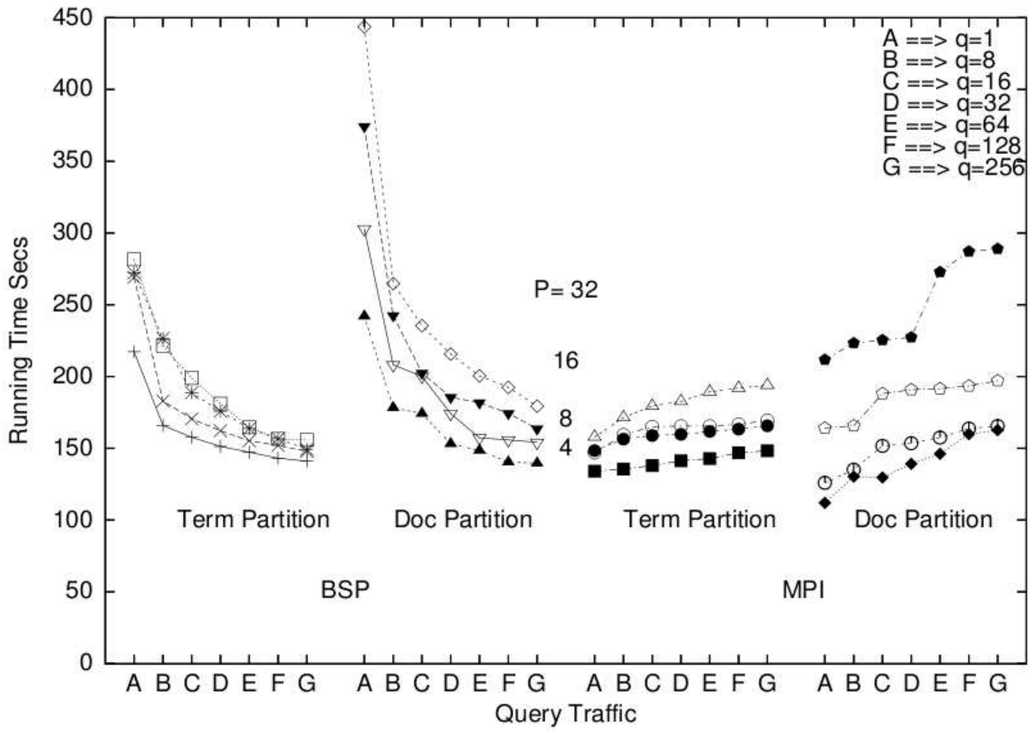
Experimental Results



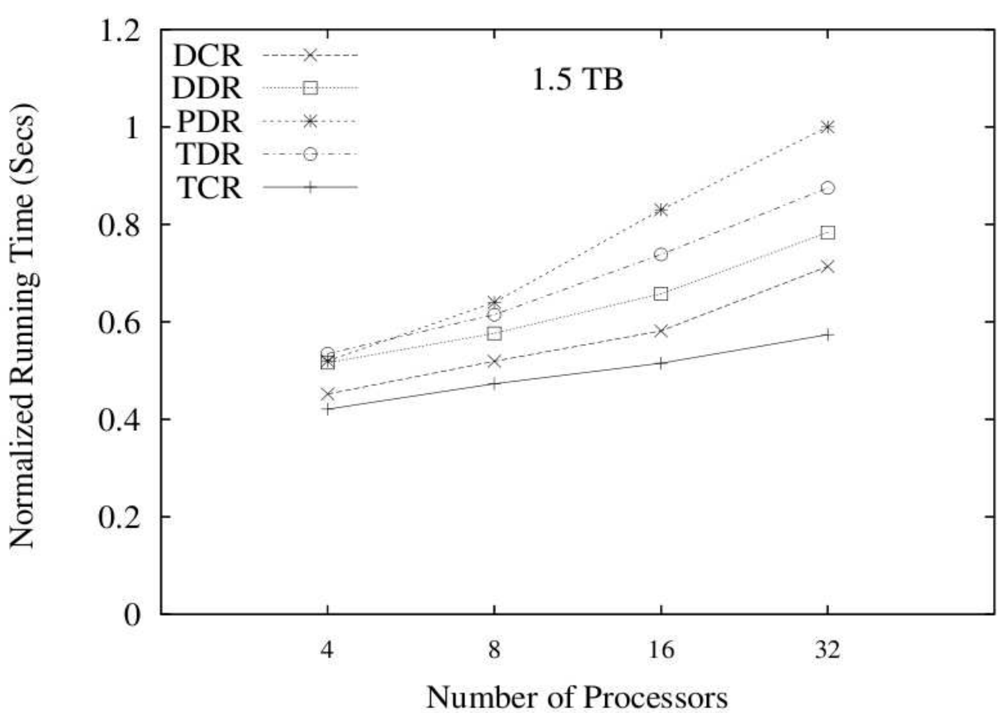
Model Comparison



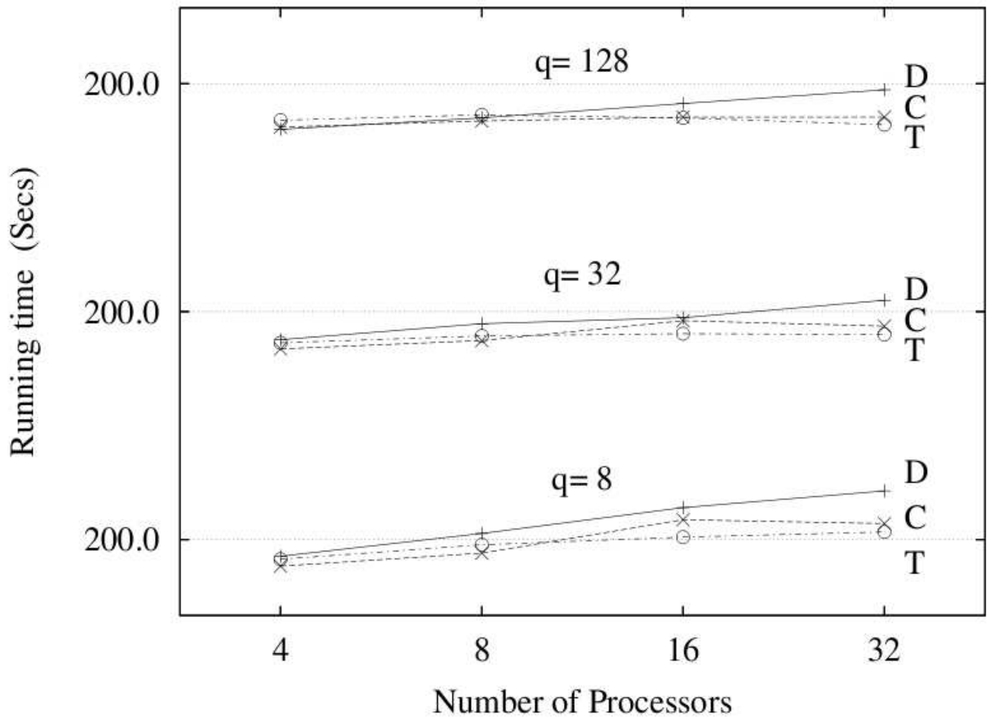
Throughput Comparison



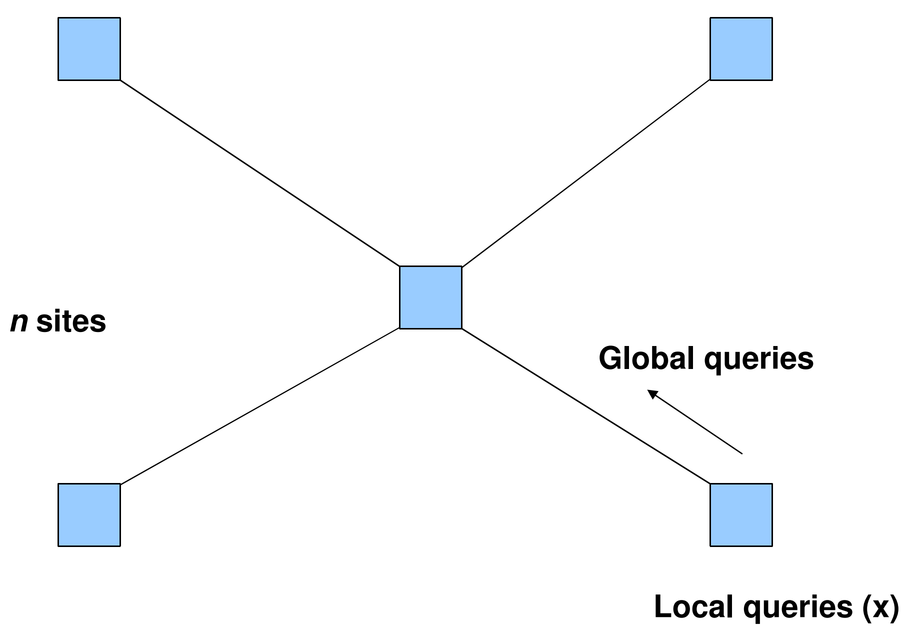
Speedup



Scalability

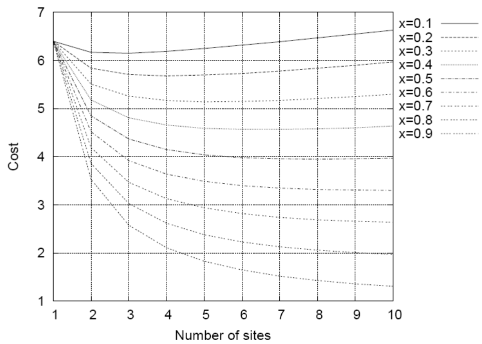


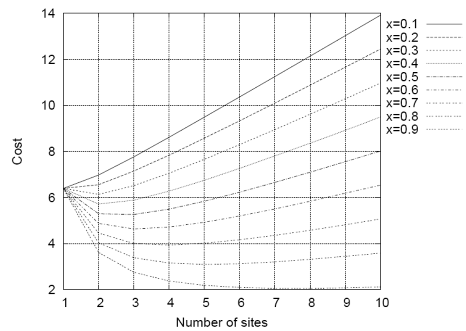
Star Topology (Baeza-Yates et al, 2008)



Cost Model

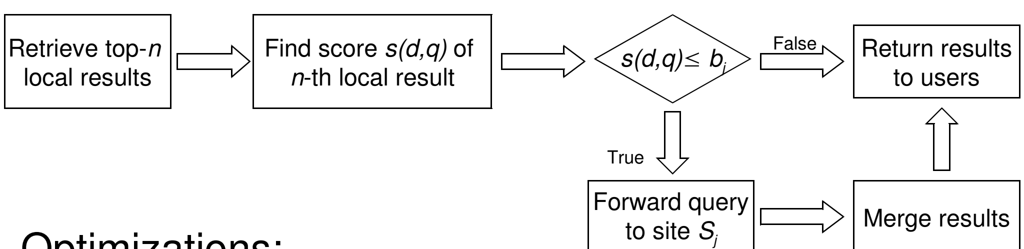
- Cost depends on **Initial cost**, **Cost of Ownership over time**, and **Bandwidth over time**.
- Cost of one QPS
  - *n* sites, *x* percentage of queries resolved locally, and relative cost of power and bandwidth 0.1 (left) and 1 (right)





Query Processing

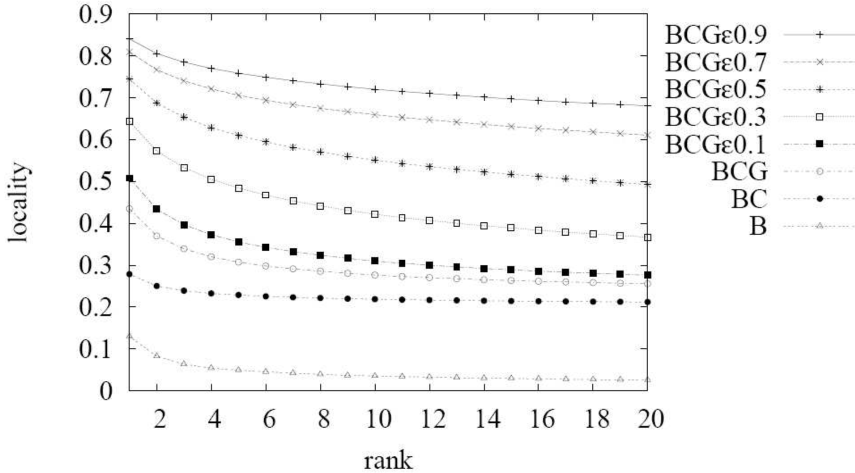
- Site S_i knows the highest possible score b_j that site S_j can return for a query
  - Assume independent query terms
- Site S_i processes query *q*:



- Optimizations:
  - Caching
  - Replication of set *G* of most frequently retrieved documents
  - Slackness factor ε replacing b_j with $(1-\varepsilon)b_j$

Query Processing Results

- Locality at rank *n* for a search engine with 5 sites
  - For what percentage of query volume, we can return top-*n* results locally



Cost Model Instantiation

- Assume a **5-site** distributed Web search engine in a **star topology**
- Optimal choice of central site S_x : site with **highest traffic** in our experiments
- Cost of distributed search engine relative to cost of centralized one

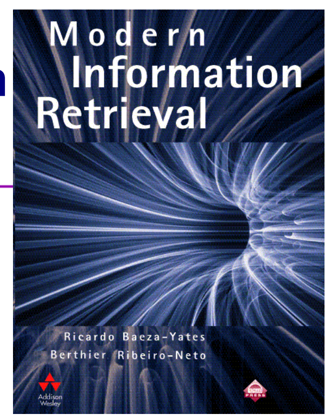
Query Processing	*Power Cost*	*Bandwidth Cost*	*Cost of distributed* / *Cost of centralized*
B	1.421	0.056	1.477
BC	1.254	0.046	1.300
BCG	1.131	0.040	1.171
BCG $_{\varepsilon}$ 0.1	1.078	0.036	1.114
BCG $_{\varepsilon}$ 0.3	0.945	0.028	**0.973**
BCG $_{\varepsilon}$ 0.5	0.807	0.020	**0.827**
BCG $_{\varepsilon}$ 0.7	0.698	0.014	**0.712**
BCG $_{\varepsilon}$ 0.9	0.634	0.011	**0.645**

Conclusions

- By using caching (mainly static) we can increase locality
- With enough locality we may have a cheaper search engine without penalizing the quality of the results or the response time
- We can predict when the next distributed level will be used to improve the response time without increasing too much the cost of the search engine
- We are currently exploring all these trade-offs

Thank you!

Second edition coming soon



Questions?

rbaeza@acm.org